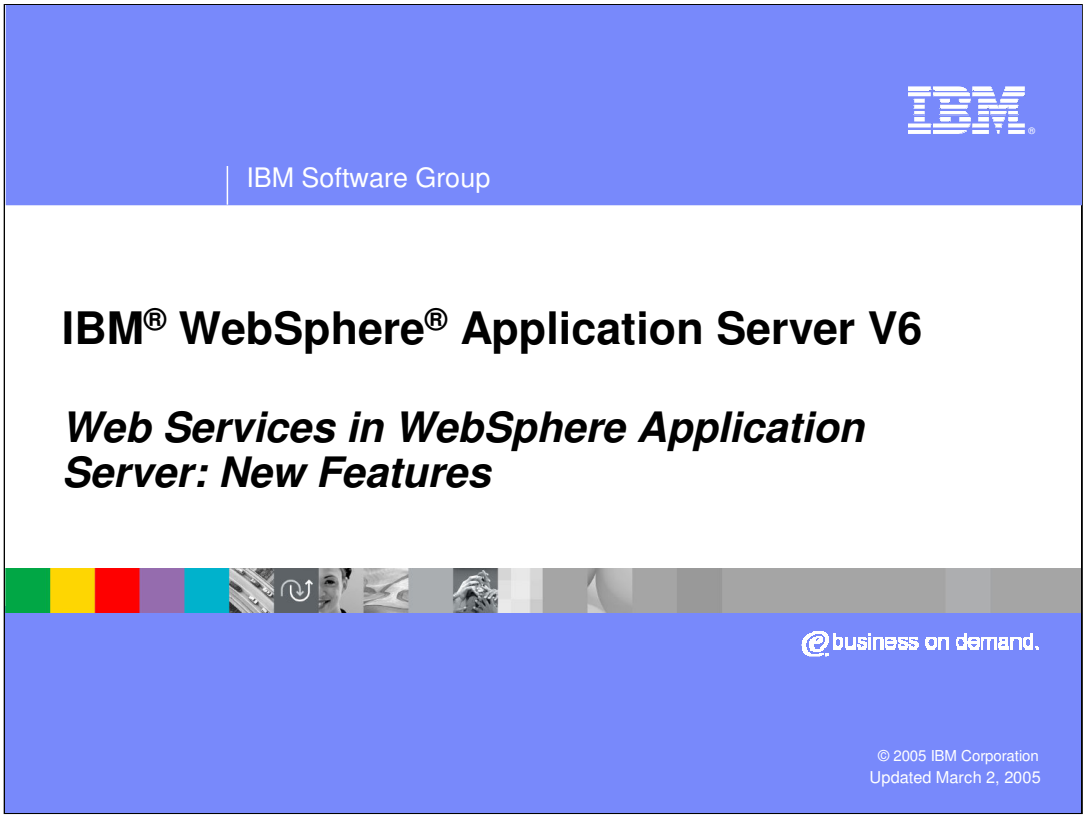


This presentation will go into detail on a number of enhancements for Web Services that are offered in WebSphere Application Server V6.

Goals

- Provide Details of
  - New Web Services functions in V6
- Prerequisite:
  - Understanding of WebSphere Application Server implementation of Web Services

This presentation will focus on the new enhancements to Web Services offered by WebSphere Application Server V6. Other presentations cover the basics of Web Services, and how J2EE Web Services work within WebSphere Application Server. Most of these new features focus on the inner workings of the Web Services Engine and Performance enhancements specific to Web Services running on WebSphere Application Server. They are not intended to be used in all Web Services deployments.

Agenda

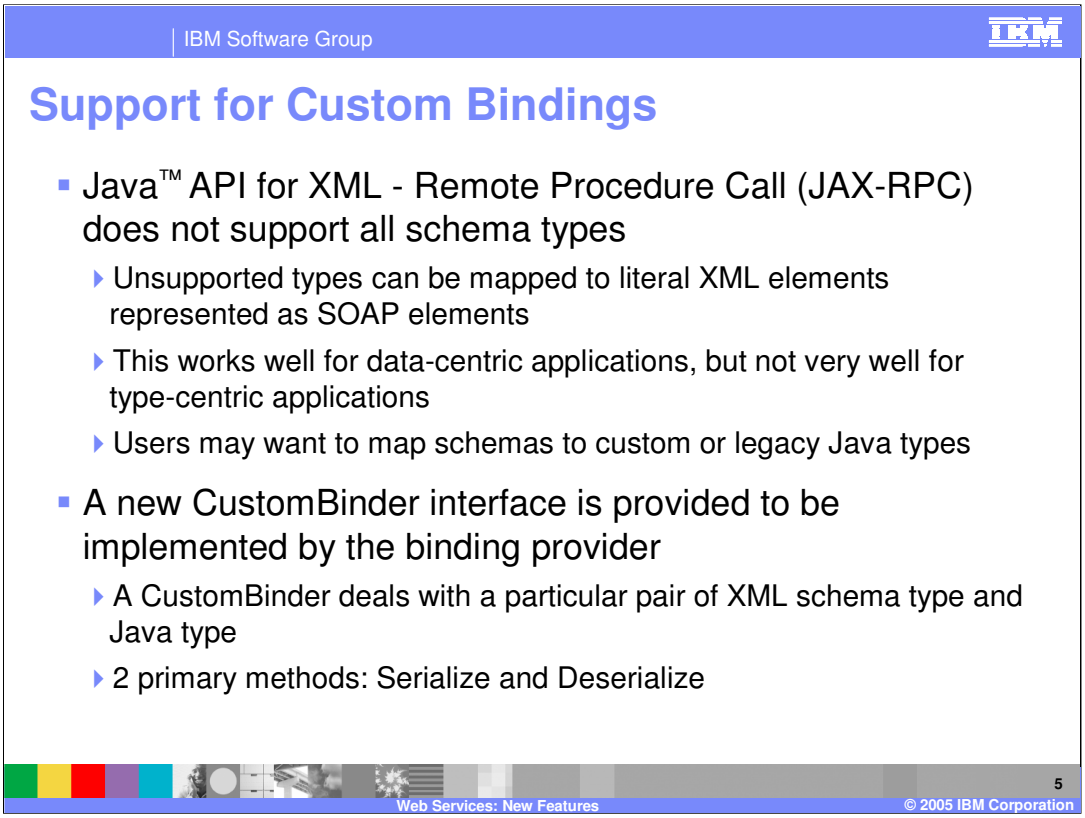
- Custom Bindings
- Support for generic SOAP elements
- Client Caching
- Multi-Protocol support

This presentation will begin by talking about the ability to define custom data bindings to be used during serialization and deserialization. Then it will explain a new capability to turn off the deserialization process for certain SOAP messages. This will cause the objects to be passed to the target service as generic SOAP elements. Next it will discuss a performance enhancement made to DynaCache that focuses on Web Services. This enhancement was actually first offered in V5.1.1 but is included here because many people were not aware of the change. Finally the last new feature that will be shown is the added support for the RMI-IIOP protocol that has been added to the JAX-RPC APIs. This enhancement allows for a better performing method to call EJB Web Services.

Section

Support for Custom Bindings

Now for an explanation of the added support for Custom Bindings.

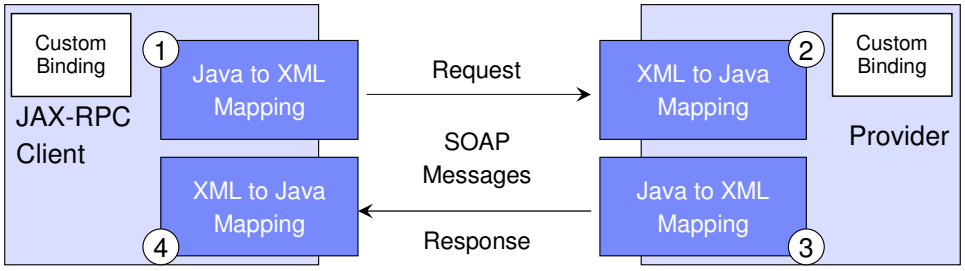


JAX-RPC specifies binding XML types in SOAP messages to Java types used by J2EE web services. The data bindings specified are limited by the specification, and may cause problems when trying to support legacy or custom data types. Using another new feature in V6, it is possible to not bind un-specified types and instead to return them as generic SOAP elements. More will be explained about that in the next section. Custom Bindings are primarily aimed at data-centric applications, that are comfortable handling XML. For type-centric applications it would be better to be able to define custom bindings, allowing applications to extend the JAX-RPC specification when needed. By extending the JAX-RPC specification this feature will limit the interoperability of a Web Service.

In order to define a custom data binding, developers would use the new CustomBinder interface. A CustomBinder deals with mapping a specific XML type to a Java type, by defining a serialize and deserialize method.

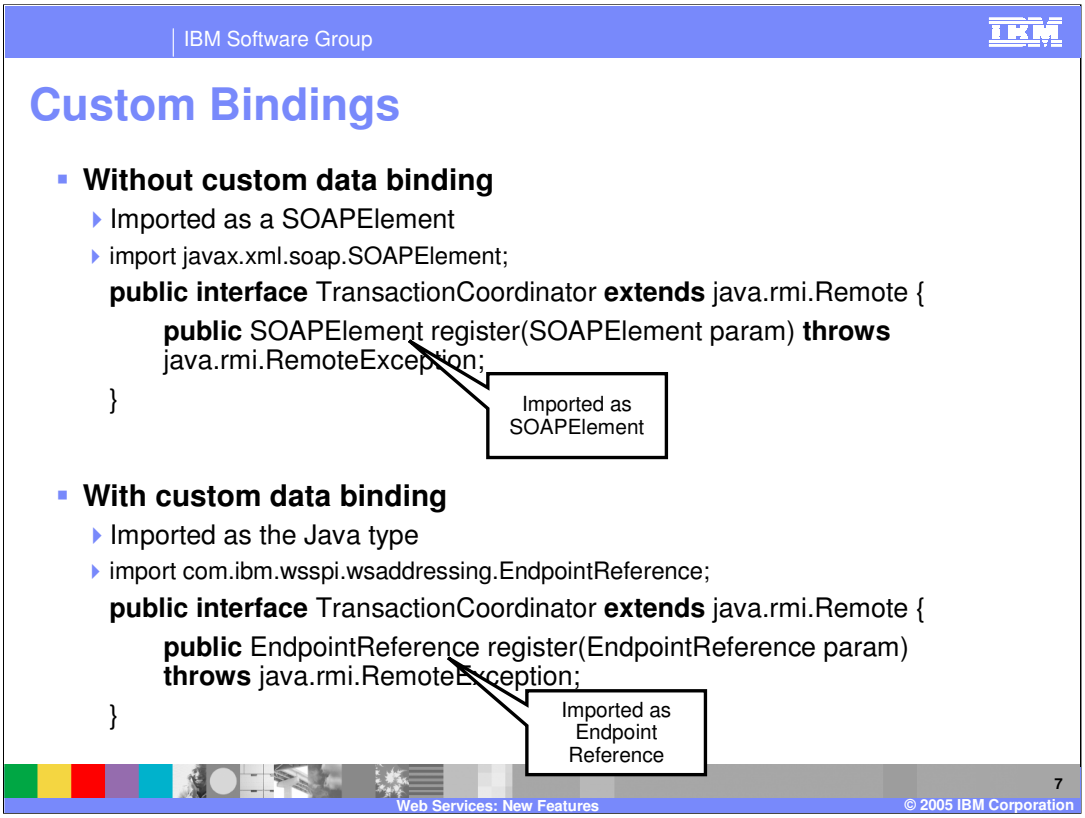
Support for Custom Bindings: Example

- JAX-RPC binds XML data to specific Java types
  - Certain designs may want to override this automatic choice
- Custom Bindings allow the developer to choose the mapping rules
- The SOAP message must use the Literal style
  - Document/Literal or RPC/Literal

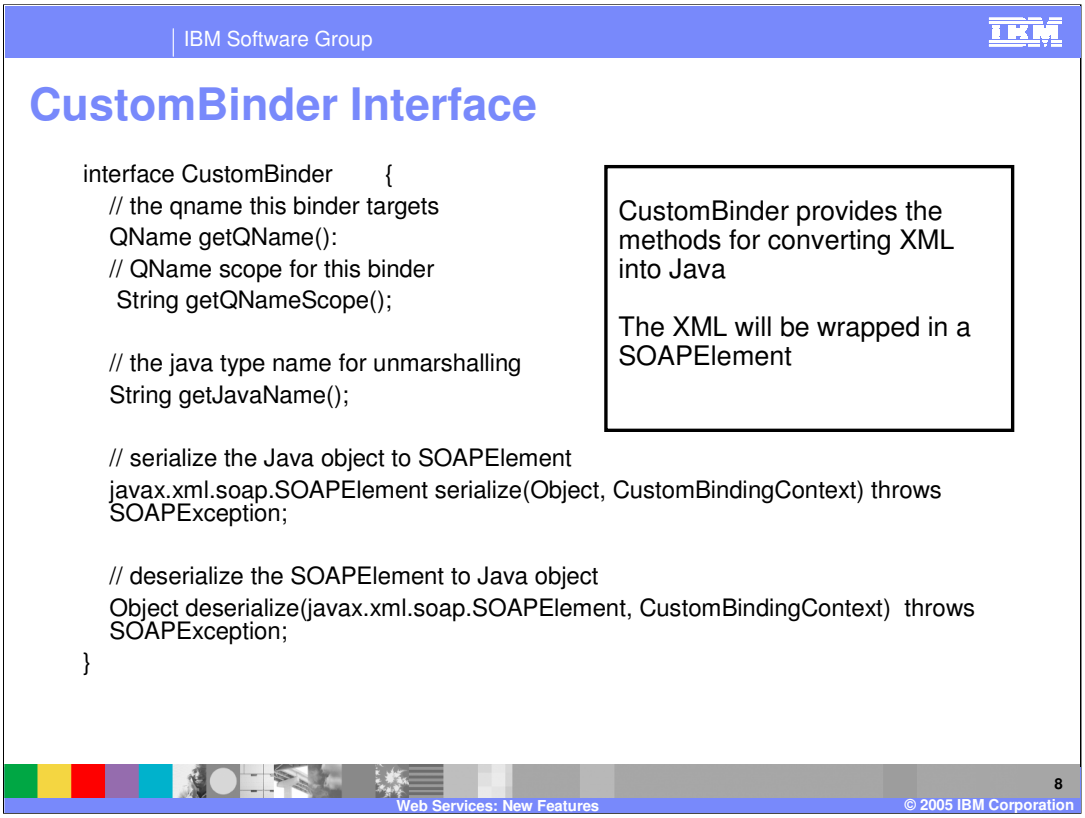


Here is an example of how a custom data binding would be implemented within a Web Service application. For a Web Service to use custom bindings the SOAP message must be sent using the Literal format, this feature does not support the encoded format. This is not much of a limitation as Document/Literal is becoming the most accepted format for sending SOAP messages, as it is supported by the Web Services Interoperability standards and provides the best performance.

In this example a Custom Binding has been created using the CustomBinder interface. This custom binding will be needed within both the client and provider application. This limits the ability to implement this feature in a solution where clients are generated by the customers. There has to be some process for getting the custom serializer to the client application, and that client must be running within WebSphere to support this feature. The custom binding will be used by the serialization engine to map the Java type to XML and vice versa, using the custom serialize and deserialize methods it contains.



Here are two examples: one using a custom data binding and the other without a custom data binding. In the top example without a custom data binding the type is imported as a SOAP element. Whereas in the bottom example using a custom binding for an endpoint reference, the parameter is imported as the proper Java type. Just as it would be if the data type was supported by the JAX-RPC specification.



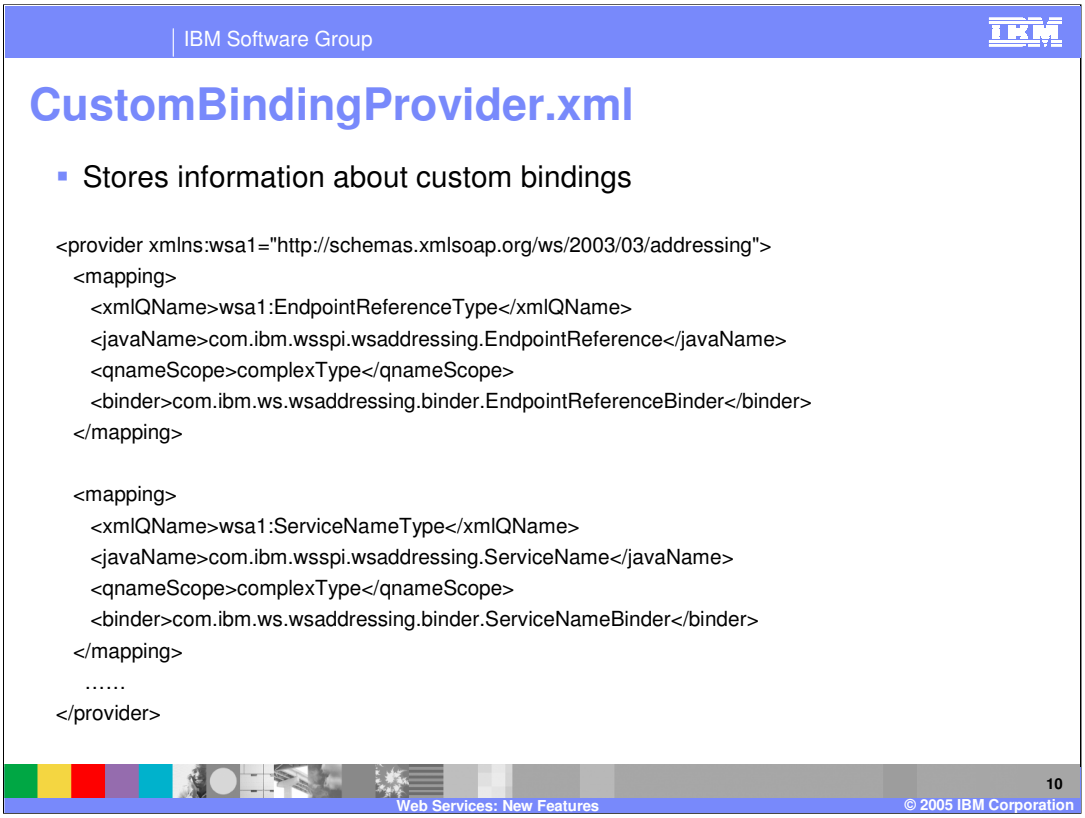
On this slide is an example of the CustomBinder interface used for defining the mapping between an XML and Java type. The Custom Binding specifies a Qname and Qname scope for the binding, the Qname scope specifies whether the binding deals with either a named or anonymous XML type. Otherwise 2 methods for mapping the Java data type to XML and mapping the XML back to Java need to be written. The XML must be wrapped in a SOAP element within these methods. This is how the parameter will be received from the SOAP message.

QNameScope

- The CustomBinder interface contains an attribute qNameScope
  - Indicates whether the binder deals with the named type or the anonymous type in the XML
  - The value for qnameScope is ‘element’ for the anonymous types
  - Or a value of ‘complexType’ or ‘simpleType’ for named types
- To combine a number of custom bindings to support a custom application a Custom Binding Provider can be used
  - Normally created for a specific custom schema file which requires the custom data binding
  - Declared in the /META-INF/services/CustomBindingProvider.XML file
  - Provided as part of the jar file

The Qname scope, specifies whether the custom binding refers to the named or anonymous XML type. The QName scope will be element for anonymous types or it will be complexType or simpleType for named XML types.

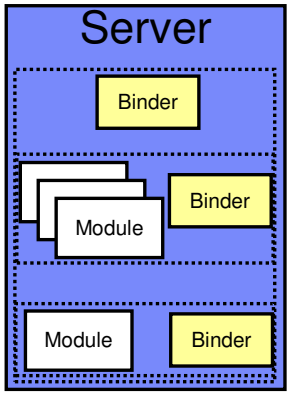
Custom Data bindings are defined to the Application Server in the Custom Binding Provider file. This Custom Binding Provider can also be used to group a number of bindings together. Groupings would be defined in the /META-INF/services/customBindingProvider.xml file and packaged as part of the jar file for the application.



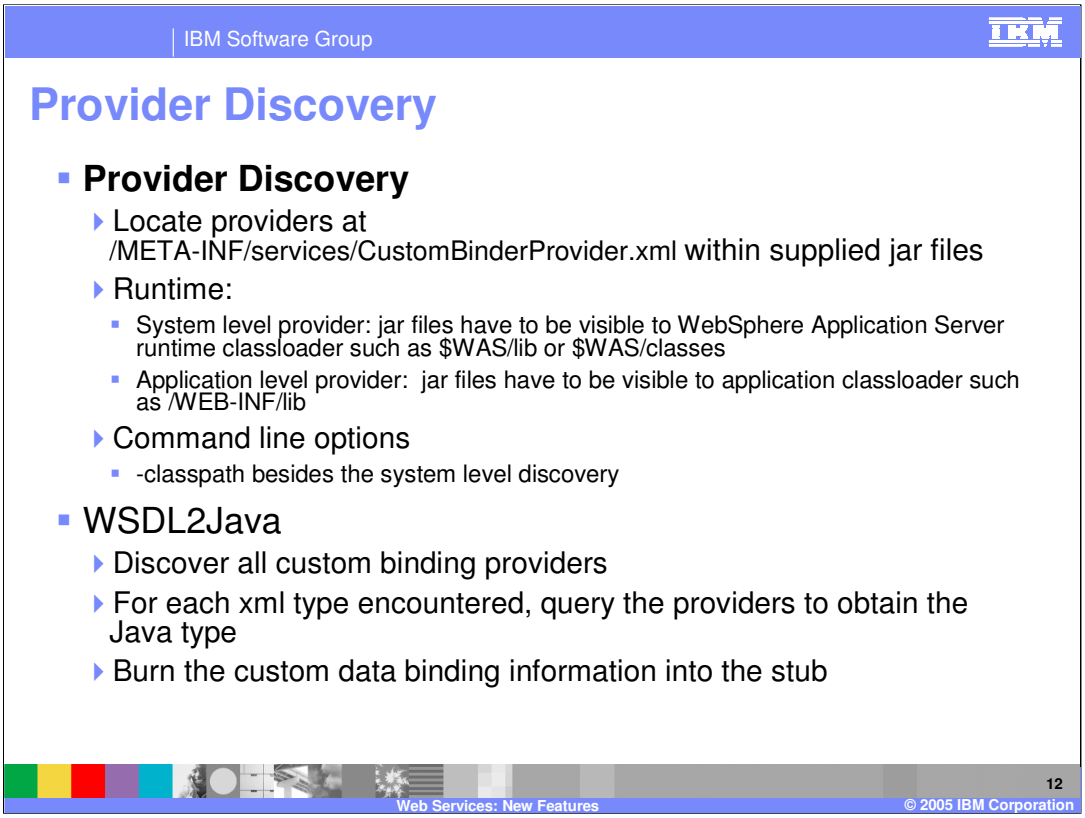
This slide shows an example of the CustomBindingProvider.xml file. This file contains all the information needed by the runtime to access the Custom Bindings that have been written. In this example there are 2 custom bindings, one for an end point reference and one for a service name. This file specifies the XML types, and Java types associated with each binder as well as which binder for the application to use when it encounters one of these types.

Binding Levels

- Depending on where the Custom Binding Provider is located it has different levels of visibility
  - Custom Binding Provider is packaged in a jar file
- Packaged with an application module
  - Visible only to the module
- Configured as part of a shared library
  - Visible to any module sharing the library
- Placed in the system directory
  - Visible to the entire runtime



Depending on where the jar file, containing the custom binding provider, is placed it will have different levels of visibility within the Application Server. This leaves 3 choices for developers. The jar file can be packaged as part of an application module, making it only visible to that application. It could be added to a shared library, so that any module that can access the library has access. Or it can also be placed within the WebSphere Application Server system directory making it visible to the entire WebSphere Application Server runtime. Developers can choose the appropriate level of access for their environment.



Based on where the Custom Binding Provider file is located the Runtime is going to use the file to find the custom bindings the Application Server will need to use. For an application level provider, the file needs to be visible to the application class loader. For a system level provider, the file needs to be visible to the runtime class loader. WSDL2Java will query the providers to obtain the appropriate Java type and burn this into the stub defining the Web Service.

Section

Support for Generic SOAP Elements

Next a feature offering support for disabling the normal deserialization process will be explained.

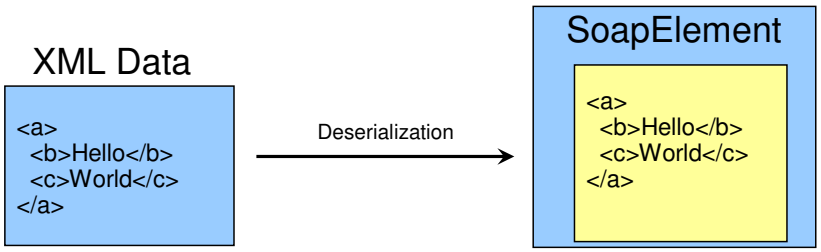
Support for Generic SOAP Elements

- In normal JAX-RPC flows, SOAP elements are deserialized into Java data types
- Support For Custom Bindings allows for the specified JAX-RPC bindings to be extended or changed
- Certain designs may prefer to eliminate the binding completely
  - Support for generic SOAP elements gives the developer the ability to disable the normal deserialization process
  - Improves performance of Services that do not require their XML bound to Java types

The process by which XML types in SOAP messages are mapped to Java types within the Java runtime has been explained a number of times by now. JAX-RPC has a list of data types it supports and the new Custom Binder feature allows developers to extend that list. But there can also be times, when developers want to completely avoid the deserialization process completely. Certain services may prefer to work directly with the XML of the SOAP message. For these types of services any type of deserialization requires them to pay performance costs they would rather avoid. This enhancement allows developers to tell the run time not to map the XML to Java types but instead to pass the target Web Service the raw XML types in the SOAP message.

Generic Elements: Example

- WSDL2Java contains a new option –noDataBinding
  - Disables normal deserialization, instead all objects will be returned bound to the SAAJ SOAPElement API
- The SAAJ API will provide a new method for accessing the xml string that represents the SOAPElement tree
  - Public final String toXMLString(…)

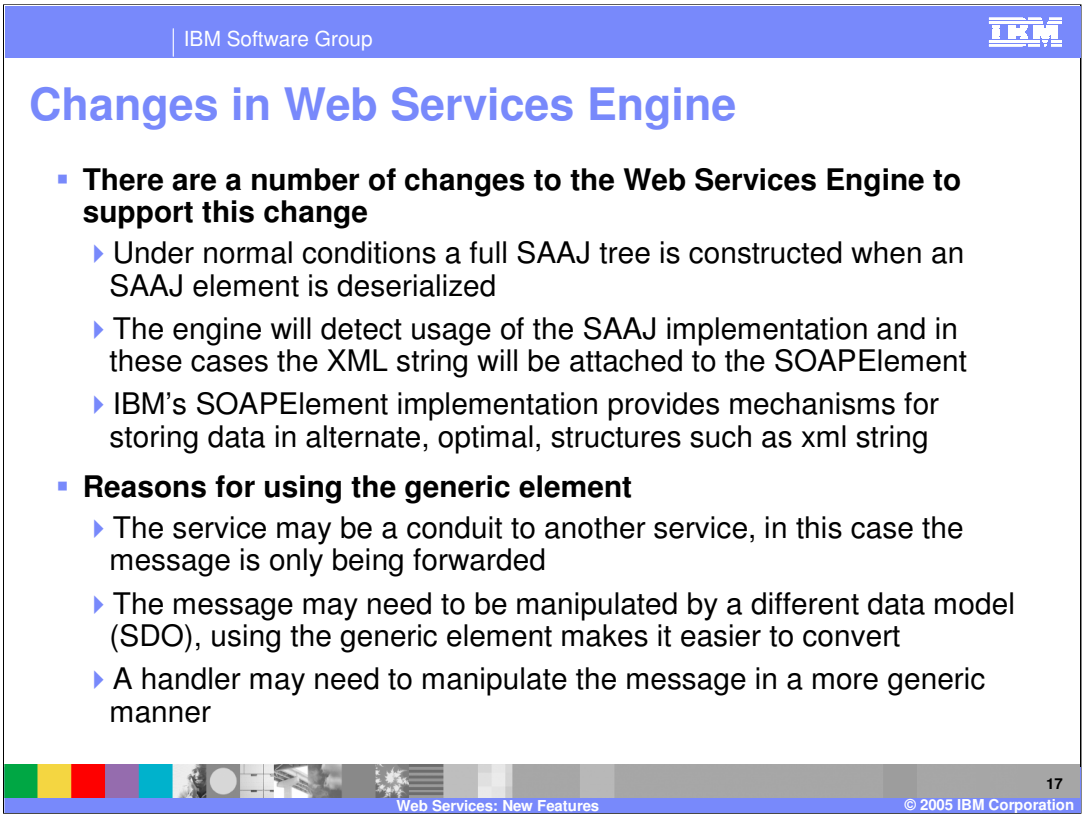


To support this change, WSDL2Java has a new option, noDataBinding. By using this option, the parameters for the Web Service will not be deserialized into Java types, but will instead be passed to the target service as a SOAP Element. The SAAJ API will provide a new method that allows developers to retrieve the XML string from the SOAP element. The code in the Web Service will have to be written to handle the XML types as appropriate. This feature allows developers to take control of this process, rather than leave it to the Java runtime.

Generic Elements: Binding Example

- Normal binding:
  - public Bean echo(Bean bean);
- Generic binding:
  - public SOAPElement echo(SOAPElement bean);
  - the SEI exposes the SAAJ interface
  - Use the toXMLString() method to retrieve the xml content from the SOAP element

Here are two examples of Web Services, the first one using a regular binding, and the second one using the new noDataBinding option. The method exposed in both of these examples is a simple Echo function that returns the parameter it receives. In the first example, the parameter has been bound to type Bean, meaning that the Web Service will be passed a parameter of the appropriate Java type. The second example uses the Object type of SOAP element. This is a part of the SAAJ interface, and using the new method toXMLString that it contains, this will allow access to the contents of the SOAPElement from within the Web Service method.



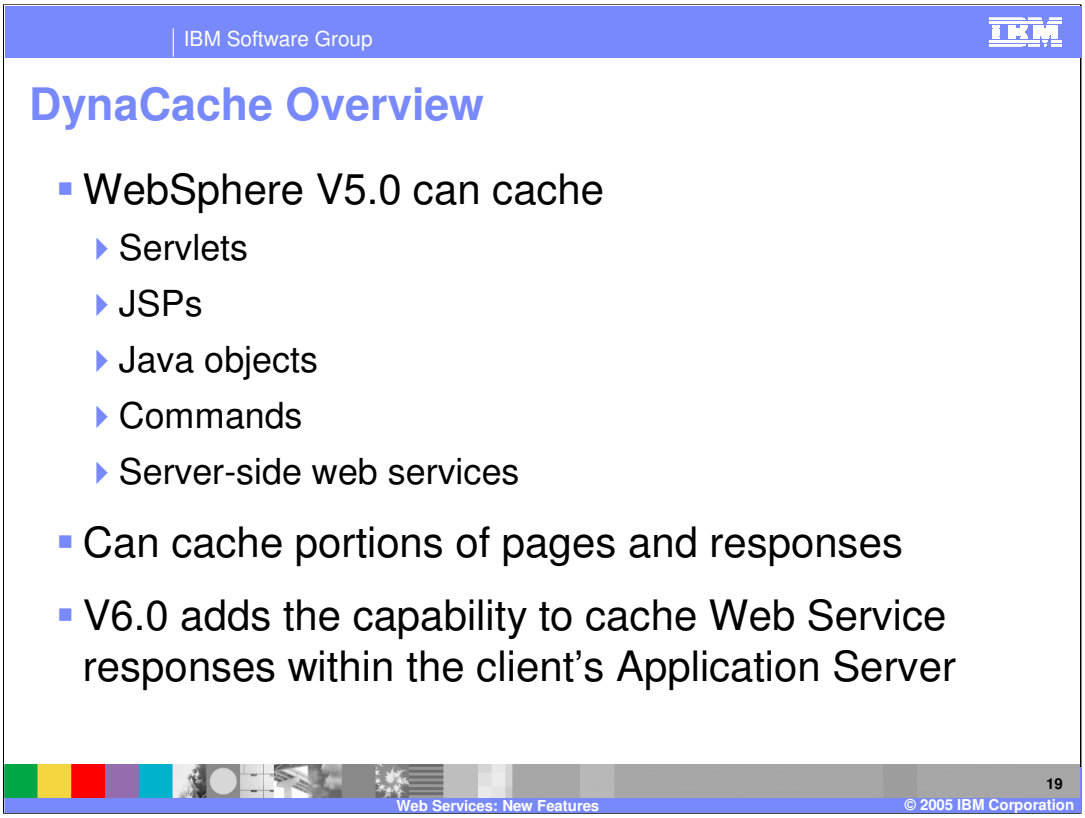
Normally the Web Services Engine will create a full SAAJ tree representing the XML of the SOAP message, when an element is deserialized. When noDataBinding is used this process is deferred, and instead only the XML string will be attached to a SOAP element. This is part of the IBM implementation of SOAP element, and allows for greater performance when dealing with the XML in the SOAP message.

Obviously there are only certain types of Web Services that would ever use this feature. The primary types of services that would want to defer deserialization are either going to forward the XML on to another service as part of a business process. Also, certain web services may prefer to deal with XML instead of Java types. Using this feature in either of these circumstances will make it easier to create these types of Web Services and help make them perform better. However, it should be mentioned that this change is aimed at a limited section of the development community.

Section

Web Services Client Caching

This is a explanation of a performance enhancement for Web Services provided by DynaCache.



DynaCache is a dynamic caching solution integrated with WebSphere Application Server to help improve performance of certain applications. Since WebSphere Application Server version 5.0 Dynacache has had the ability to cache full responses, or portions of a response to calls made to Servlets, JSPs, Plain Old Java Objects, and server side Web Services. Version 6 adds the capability to cache responses to Web Services within the DynaCache on the clients Application Server. This will further increase performance of Web Services clients that run within a WebSphere Application Server.

Client Caching

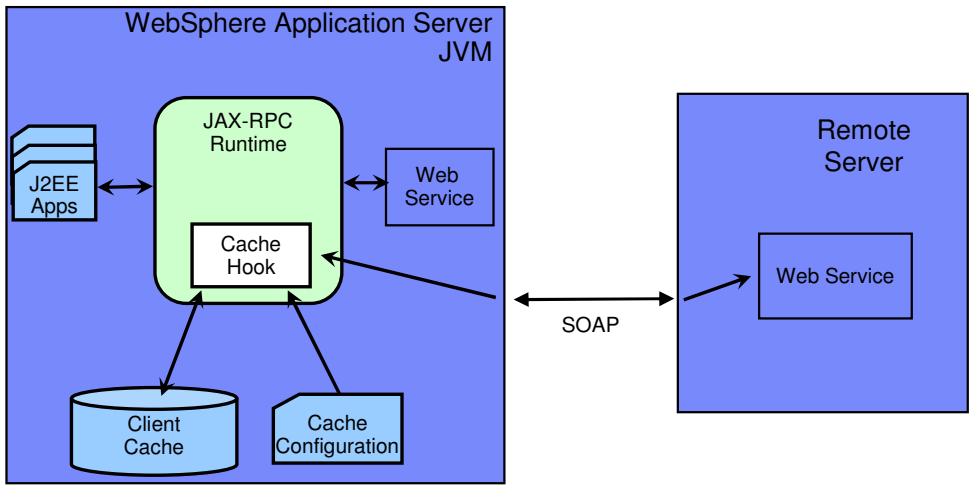
- Increases the performance of Web Services clients by caching responses from remote Web Services
  - Once a response is cached, subsequent calls to the same Web Service with the same set of request parameters could be responded from cache
- Provided as a JAX-RPC handler
  - Based on the policy specified in the cachespec.xml file
- Choice of methods to invalidate cached values
  - Rule Based, Time Based, APIs

DynaCache increases performance by responding to requests from a value stored in cache rather than invoking the actual service or code being called. Once a response has been added to the cache, any matching calls made to the service can be responded to from the cached value, so long as the value in the cache is considered valid.

There are a number of ways to invalidate a value once it has been placed into the cache. Rules can be set within Dynacache, though this is the least likely to be used with web services. Time based invalidation, will invalidate an entry after a certain amount of time has passed. It is also possible to invalidate an entry in cache through code.

The new client caching capabilities are provided via a JAX-RPC handler. This handler uses the caching policy specified in the CacheSpec.xml file. This file contains all dynacache caching policies and is not specific to just the new Client Caching feature.

Architecture: Big Picture



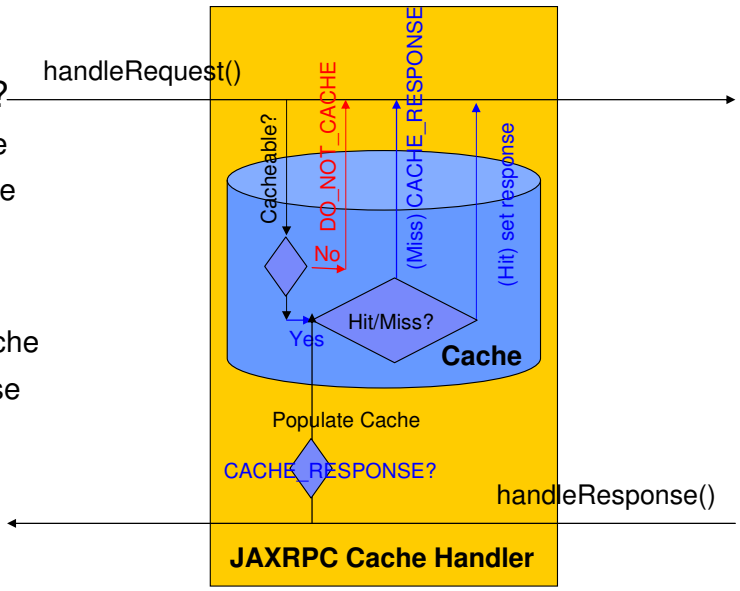
A small performance penalty is paid to check the cache policy on each invocation

Here is an example of how the JAX-RPC client cache handler fits within the Application Server's Java Virtual Machine or JVM.

Within the JVM the JAX-RPC runtime has a hook to the caching service. When a client request comes into the RPC runtime, it is intercepted by the cache handler that checks the cache based on rules found in the cache configuration XML file. If it doesn't find the information in the Cache, then it will either call the Web Service within the same WebSphere server, or forward the call on to the target Web Service located elsewhere. This means the Web Service can be local or remote to this server. The result would be placed in the cache before being returned to the client.

JAX-RPC Cache Handler: Details

- Request
- Is it Cacheable?
  - No: do not cache
  - Yes: check cache
- Does it exist in Cache?
  - No: populate cache
  - Yes: set response
- Response



This slide is a more detailed look at the choices being performed within the Cache Handler.

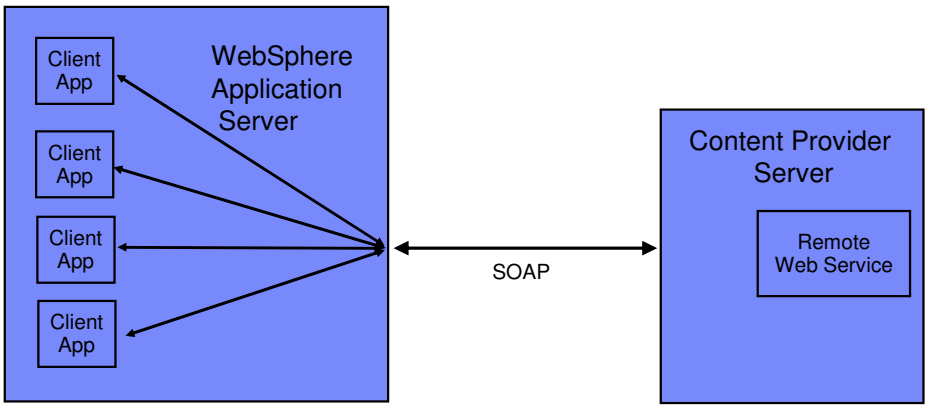
Web services client caching is provided as a JAX-RPC cache handler. In the handleRequest() method, cache configuration manager is searched for a cache policy based on the target endpoint address specified in the request. Request is not cacheable if a matching cache policy is not found. If a matching policy is found, all the cache id rules in that policy are executed one by one until a valid rule is identified. Result of the first valid cache rule will be the cache key for lookup. If this lookup ends in a cache miss, a property is added to the handler chain's message context to cache the response in handleResponse() method. If this lookup ends in a cache hit, the value from the cache is set as the response and the rest of the request handle chain is blocked. If a SOAP fault is returned, the response is not cached. Else it will be cached in handleResponse() method using the cache key specified in the message context.

Scenarios

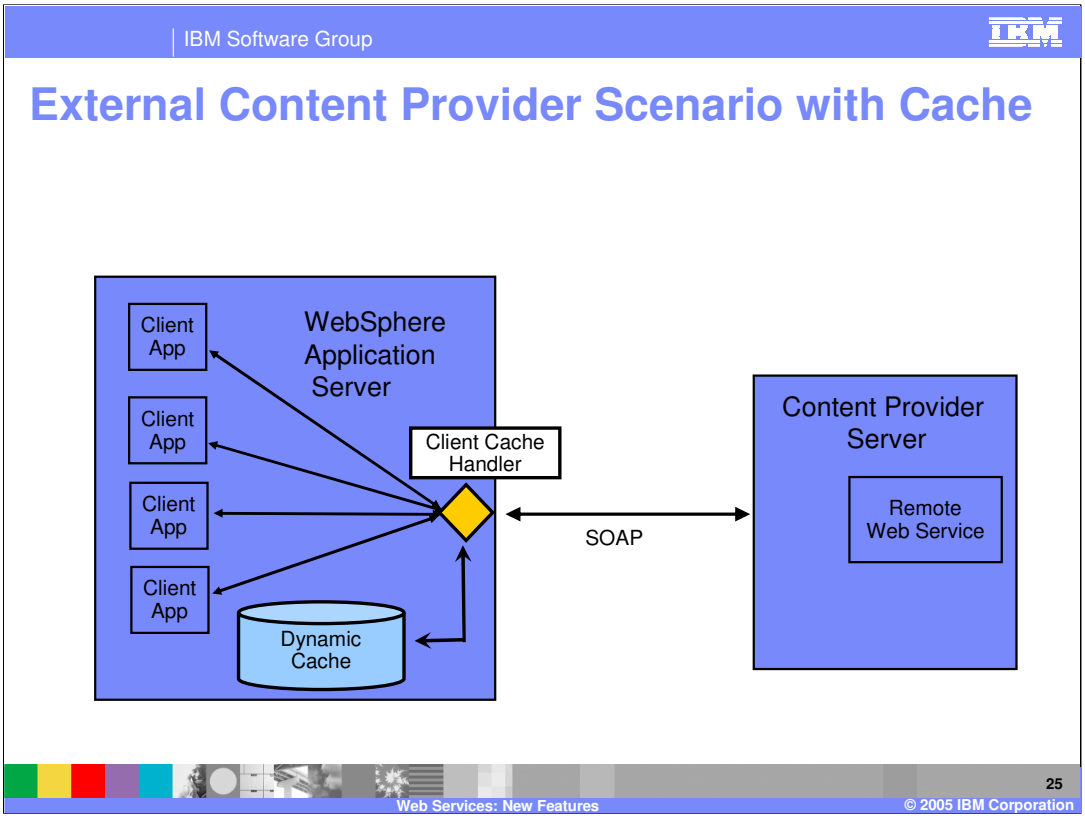
- Enterprise applications hosted on content provider's network exchanging SOAP messages
- Reverse proxy acting as a gateway by invoking Web Services
  - Proxy can respond without invoking target services
- Split-Tier setup
  - Both client and server running WebSphere Application Server

The next few slides are going to look at a number of scenarios where using a client side cache will help improve performance. These will help to illustrate the enhancements that a client side cache makes possible.

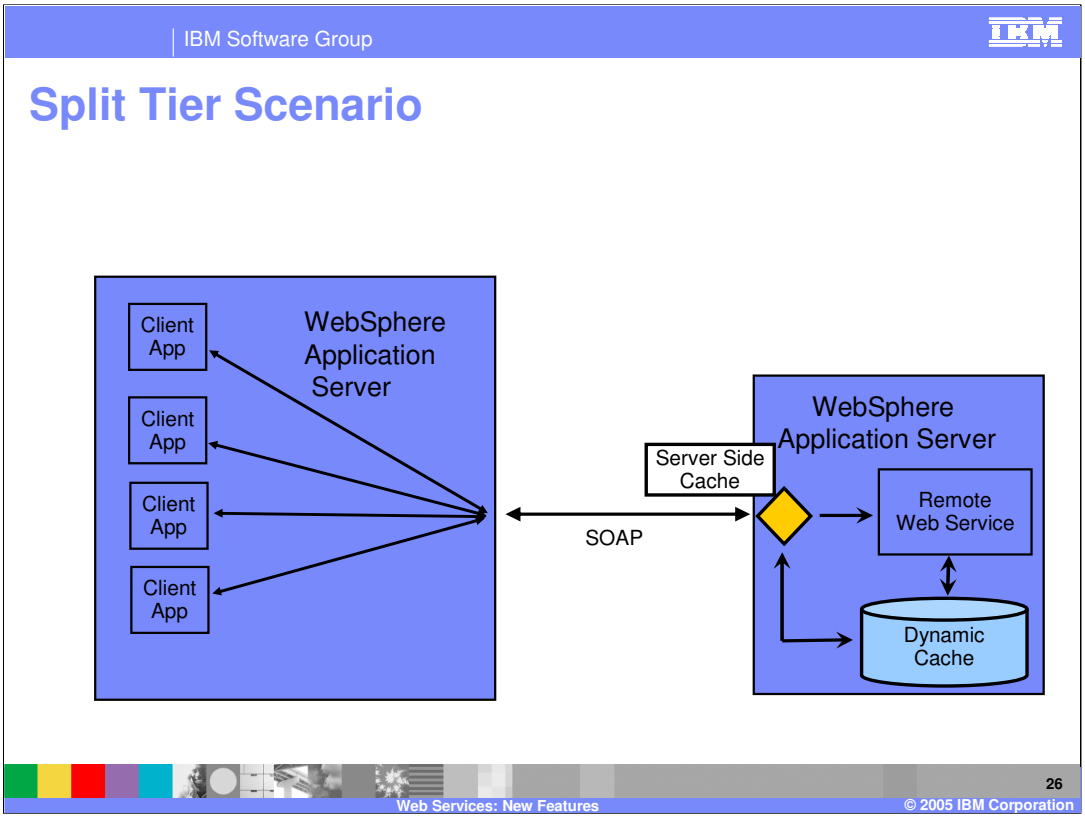
External Content Provider Scenario without Cache



In the first example there is an existing Web Services scenario, with multiple Web Services clients running within a WebSphere Application Server. They access a Web Services provider by sending SOAP messages over the internet. In this example each separate call must be made to the target provider, depending on the number of repetitive calls this can get expensive from a performance perspective.

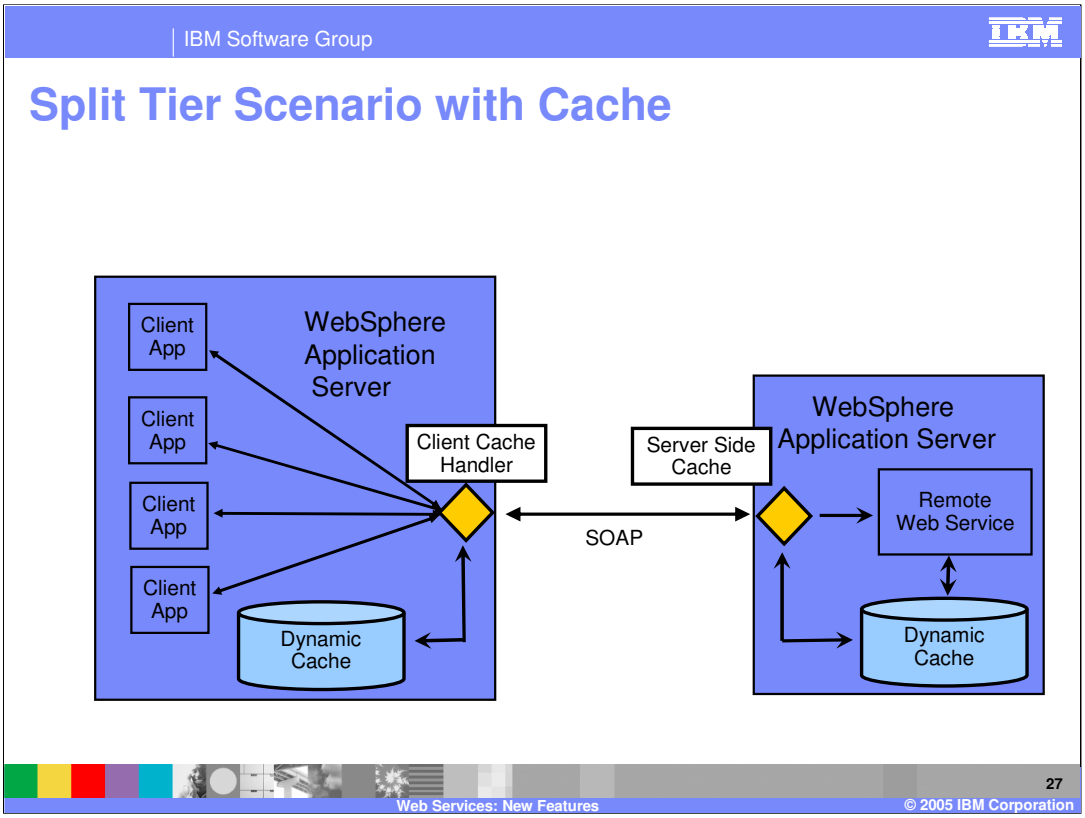


Now take that previous example and add a client side cache. Again this is simply a JAX-RPC handler that will interact with the DynaCache within the Client's Application Server. This allows each of the Web Services clients to cache their results most likely for some specific period of time. Repetitive calls made within that time frame will be responded from cache, preventing unnecessary calls over the internet. When those cache values become stale, the next call made to the Web Service will be made as normal, and that value would repopulate the cache value. This can greatly lower the number of Web Services calls being made in certain scenarios.



The next example deals with a split tier environment, in which a reverse proxy server is sending client requests to a service provider within the same intranet. The Reverse proxy server would most likely be a Web Services Gateway or similar application in this scenario.

The Web Services provider in this case is already using DynaCache's capabilities. From a provider perspective, these features have existed since V5. This will increase performance on the provider side, by responding to some requests via cache, instead of running the Web Services provider code. However this scenario can be improved.



Take that previous example and add a client side cache to the reverse proxy server and now the proxy server can respond to requests without having to call the actual services provider. Also the Web Services provider can still use dynacache to further enhance performance. Since the client side cache is represented as a JAX-RPC handler it can be installed into the Web Services Gateway, improving performance by minimizing calls to the actual service providers.

Enabling JAX-RPC Cache

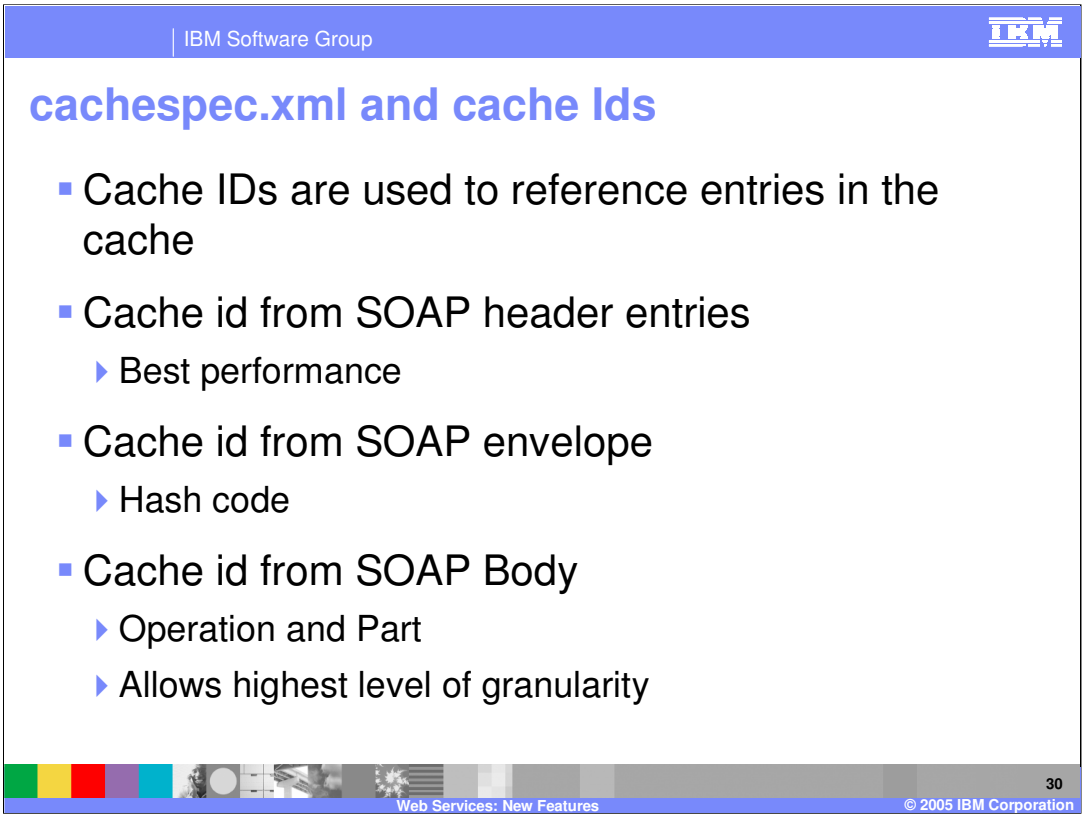
- JAX-RPC caching is enabled if Dynamic cache Service is enabled
- Configure caching policy in cachespec.xml
  - New type of configuration entry “JAXRPCClient
  - Supporting new types “part”, “operation”
- The cachespec.xml file is found inside the WEB-INF directory of a Web module

Turning on the Client Caching capability is as simple as enabling DynaCache within the administration console. In the case of a Services Gateway, DynaCache would have to be enabled on the Application Server the gateway application is installed.

The caching policy for the handler is specified in the cache spec XML file. There is a new type of entry that can be placed in this policy file called JAXRPCClient to support these changes. This file can be global, if it is located in the Application Server properties directory. Though, the recommended method is to keep the xml file with the deployment module of your application.

Caching Example

Now for an example of implementing client side caching.



When data is cached in WebSphere a cache ID is created to help store and retrieve the information in the cache. WebSphere has a number of choices when creating a cache id, depending on the level of granularity needed. The ID can be created from the SOAP header. This is the best performing method for creating the ID, but it is also the least granular. Otherwise the ID can be generated from the SOAP envelope or from entries in the SOAP body. These allow more control over which data is stored in the cache, but they can also be more difficult to set up.

Sample WSDL

```
<definitions targetNamespace=http://TradeSample.com/…>
    <message name="getQuoteRequest">
        <part name="symbol" type="xsd:string"/>
    </message>
    .....
    .....
    <binding name="SoapBinding" type="tns:GetQuote">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
            <operation name="getQuote">
                <soap:operation soapAction=""/>
                <input name="getQuoteRequest">
                    <soap:body namespace="http://TradeSample.com/" use="encoded"
                    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
                </input>
                ......
                </operation>
    </binding>
    <service name="GetQuoteService">
        <port binding="tns:SoapBinding" name="SoapPort">
            <soap:address location="http://TradeSample.com:9080/service/getquote"/>
        </port>
    </service>
</definitions>
```


Here you see portions of an example WSDL for a stock quote service. It contains a method for getQuote, which requires a parameter ‘symbol’ which would be the stock name like IBM. The various bolded areas are information you would need for cache ID’s

Sample SOAP Request

```
POST /wsgwsoap1/soaprpcrouther HTTP/1.1
….
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope …>
<soapenv:Header>
  <getQuote soapenv:actor="com.ibm.websphere.cache">
    IBM
  </getQuote>
</soapenv:Header>
<soapenv:Body … >
  <getQuote xmlns="urn:ibmwsgw#GetQuoteSample">
    <symbol xsi:type="xsd:string">IBM</symbol>
  </getQuote>
</soapenv:Body>
</soapenv:Envelope>
```


Here is a small portion of a sample WSDL that will be used for the example. The WSDL is exposing a method called getQuote. This method takes a parameter of symbol, representing the stock symbol for a company. The other portion shown here is the actual address location for this service.

Cache ID from SOAP Header

```
<cache>
  <cache-entry>
    <class>JAXRPCClient</class>
    <name>http://TradeSample.com:9080/service/getquote</name>
    <cache-id>
      <component id="getQuote" type="SOAPHeaderEntry"/>
    </cache-id>
  </cache-entry>
</cache>
```

- Cache ID is **http://TradeSample.com:9080/service/getWQuote:getQuote=IBM**

In this example of a cache entry using the SOAP header to create the cache id, the cache entry class is JAXRPCClient. The name is the tradesample service getQuote binding. The cache id generated from this is shown on the bottom of the slide. So from this cache entry example a response to the getQuote method for IBM would be placed in the cache.

Cache ID from SOAP Envelope

```
<cache>
  <cache-entry>
    <class>JAXRPCClient</class>
    <name>http://TradeSample.com:9080/service/getquote</name>
    <cache-id>
      <component id="hash" type="SOAPEnvelope"/>
      <timeout>60</timeout>
    </cache-id>
  </cache-entry>
</cache>
```

- Cache ID is
  http://TradeSample.com:9080/service/getquote:Hash=<*xxxHashSoapEnvelope*>

This is an example of getting the information from the SOAP envelope. This performs slightly slower then the SOAP header example, because it requires some parsing of the SOAP message to retrieve this information. The name is the SOAP port coming in, and a HASH on the SOAP envelope is specified. The id value that is created contains the hash value for the SOAP envelope.

Cache ID from SOAP Body

```
<cache>
  <cache-entry>
    <class>JAXRPCClient</class>
    <name>http://TradeSample.com:9080/service/getquote</name>
    <cache-id>
      <component id="" type="operation">
        <value>http://TradeSample.com/:getQuote</value>
      </component>
      <component id="symbol" type="part"/>
    </cache-id>
  </cache-entry>
</cache>
```

- Cache ID is
  http://TradeSample.com:9080/service/getquote:operation=http://TradeSample.com/:getQuote/symbol=IBM

This is an example showing how to create a cache id from the SOAP body. This method allows the greatest control in selecting what is cached, but also requires the largest performance penalty as the entire XML message must be parsed by the cache handler to retrieve this information. This would allow the ability to cache certain portions of an XML message that will be common across multiple service requests.

Section

Multi-Protocol Support

Finally are a number of slides on the new multi-protocol enhancements made to JAX-RPC within WebSphere Application Server V6.0. This enhancement extends the JAX-RPC support to allow for use of RMI-IIOP calls to EJB based Web Services.

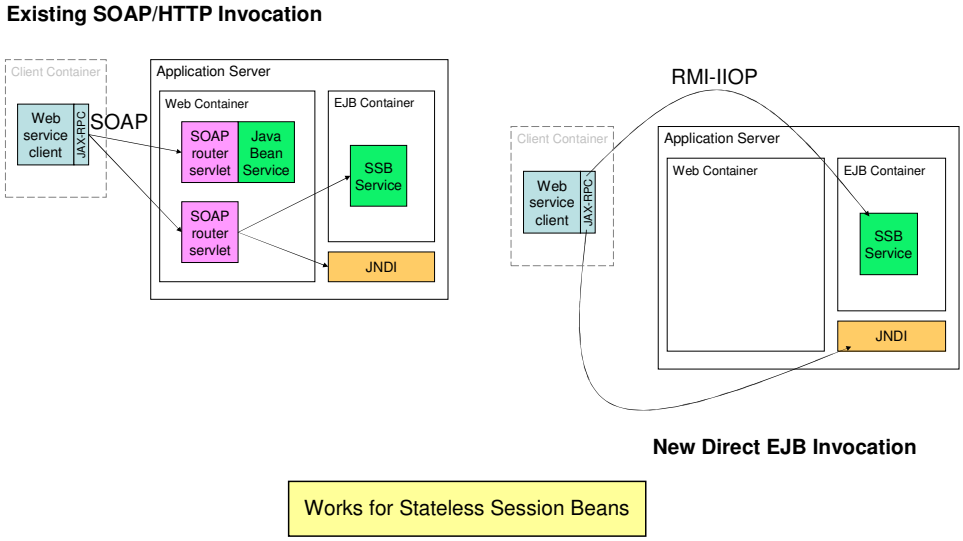
Multi Protocol Support

- Extends JAX-RPC support for invoking remote stateless session EJBs with RMI-IIOP
  - Better performing method for calling EJB services
- This allows managed clients (defined by JSR 109) to access Web Services through a number of protocols
  - No changes to JAX-RPC client are needed

With this change IBM is extending the JAX-RPC support for invoking EJB Web Services. JAX-RPC already supports SOAP over HTTP, and IBM extended that with support for JMS. With these changes Stateless Session EJBs can also be invoked using RMI-IIOP. This is the preferred method for EJB web services due to the performance gains.

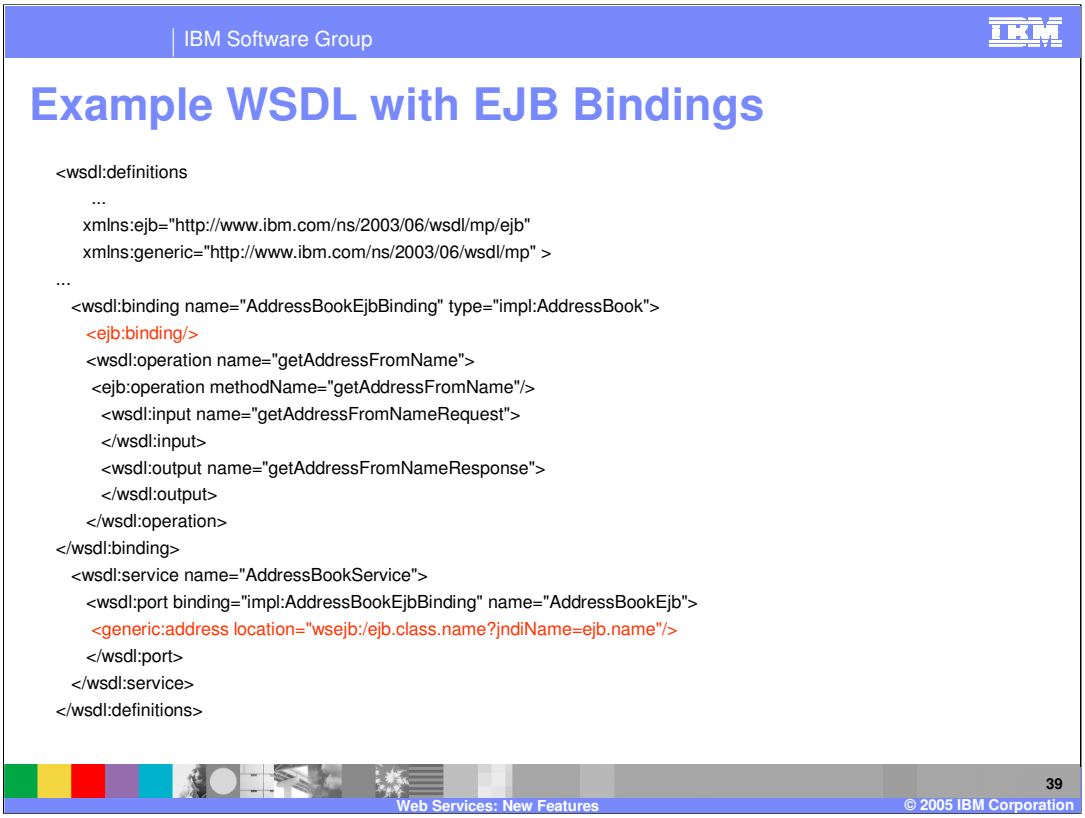
These changes do not require any changes to an existing JAX-RPC managed client. This simply adds more APIs that can used to invoke web services. A JAX-RPC client that is running within WebSphere now has the option to directly call EJB's using these new methods, these changes should be mostly transparent to the developers.

Supported Flows for Java Bean and EJB



This slide shows how a Stateless Session EJB is invoked within the WebSphere Application Server. On the upper left there is the old way of accessing EJBs. This required the client to send a SOAP message to a router servlet contained within the web container. The router servlet would then invoke the EJB service in the EJB container. The added overhead of the router servlet decreased performance for EJB web services using JAX-RPC. This led to a lot of developers using WSIF to invoke EJB services.

With these changes, the router servlet will be bypassed, and instead stateless session EJBs can be called in a more natural way from JAX-RPC clients. The client will communicate directly with the EJB service using RMI-IIOP.



These changes require some small changes in the WSDL document that will be generated for EJB services. This example shows that the binding is defined as an EJB binding. The methodName attribute is the EJB home interface method name to be invoked for the abstract operation name which encloses it in the WSDL. Lastly the address for the EJB service includes a JNDI name that will be used to look up the EJB within the target application server.

Other Changes

- WSDL2Java
  - No new command-line options
  - Changed to recognize non-SOAP ports and bindings
  - Locator and Stub classes will support non-SOAP ports
  - New Information class contains service information previously contained in the stub
- Java2WSDL
  - Changed to create non-SOAP bindings in the WSDL
  - Command-line now supports EJB bindings
    - java2WSDL –bindingTypes http,ejb –implClass my.pkg.MyEJBClass my.pkg.MySEI

Some changes are also needed in the supporting tools associated with web services. WSDL2Java has been changed to recognize non SOAP bindings. The stub classes that WSDL2Java generates will also be changed to support non SOAP ports. There has also been the inclusion of a new Information class to contain information on the service previously contained within the stub.

Java2WSDL was changed to create the new WSDL document that was shown on the last slide. When running Java2WSDL from the command line there is a new EJB option for creating bindings.

Section

Summary and Reference

Now for the summary and reference sections of the presentation.

This presentation covered a number of enhancements offered by WebSphere Application Server V6 for Web Services. These enhancements are WebSphere specific extensions to the J2EE specifications. They are very specific in their uses, and would not be appropriate in most Web Service applications. Following this Summary are a number of references to other materials that can be used to learn more about Web Services.

Resources: JSR 101, 109

- JSR 101 (JAX-RPC)
  - http://java.sun.com/xml/jaxrpc/index.html
- JSR 109
  - http://jcp.org/jsr/detail/109.jsp
  - http://www-106.ibm.com/developerworks/webservices/library/ws-jsr109/index.pdf
- Introduction to Web Services
  - http://java.sun.com/webservices/docs/ea2/tutorial/doc/IntroWS.html

Resources: IBM

- http://www.ibm.com/software/ad/studioappdev
- http://www.ibm.com/software/webservices
- http://www.ibm.com/developerworks/webservices
- http://www.alphaworks.ibm.com/webservices
- http://www.redbooks.ibm.com
  - SG246891 - WebSphere V5 Web Services Handbook
- http://www.eclipse.org

Resources: Client Caching

- WebSphere V5.1.1 Information Center
- "Caching In" – WebSphere Journal
  - http://sys-con.com/story/?storyid=44291&DE=1
- **IBM WebSphere V5.0 Performance, Scalability, and High Availability**
  - http://publib-b.boulder.ibm.com/Redbooks.nsf/RedpieceAbstracts/sg246198.html?Open
- Developing and Deploying Command Caching with WebSphere Studio V5
  - http://www-106.ibm.com/developerworks/websphere/registered/tutorials/0306_mcguinnes/mcguinnes.html

Trademarks, Copyrights, and Disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	CICS	IMS	MQSeries	Tivoli
IBM(logo)	Cloudscape	Informix	OS/390	WebSphere
e(logo)business	DB2	iSeries	OS/400	xSeries
AIX	DB2 Universal Database	Lotus	pSeries	zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds.

Other company, product and service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements and/or changes in the product(s) and/or program(s) described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (e.g., IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.